Mathematical Case Studies: Tools

R.D. Arthan
Lemma 1 Ltd.
rda@lemma-one.com

11 April 2016

Abstract

This document describes some tools that are used in the proofs in the ProofPower Mathematical Case Studies.

Reference: LEMMA1/HOL/WRK083; issue 1.115

Contents

To Do

- Think through the packaging of the tools
- Add more powerful tools.

1 INTRODUCTION

Currently the tools describes some utilities primarily for use in the code of the tools themselves and simple support for proving that an expression denotes a morphism in a concrete category finitely generated by certain given morphism constructors and object constructors.

2 UTILITIES

SML

```
fun thm_frees (thm : THM) : TERM list = (
        frees (list_mk_∧ (concl thm :: asms thm))
);
```

SML

```
new_error_message{id = 999001,
        text = "?0 is not of form ?1"};
new_error_message{id = 999002,
        text = "?0 expects a ?1-element argument list"};
new_error_message{id = 999003,
        text = "hd2 expects a list with at least 2 elements"};
fun dest_any (pattern : TERM, fun_name : string) : TERM -> TERM list = (
        let     val (f, args) = strip_app pattern;
                val arity = length args;
                fun strip_and_check tm = (
                        let     val (g, args) = strip_app tm;
                                val _ = term_match g f;
                        in      if      length args = arity
                                then    args
                                else    fail "" 0 []
                        end     handle Fail _ => (
                                term_fail fun_name 999001 [tm, pattern]
                        )
                );
        in      strip_and_check
        end
);
```

SML

```
fun is_any (pattern : TERM) : TERM -> bool = (
        let     val dest = dest_any (pattern, "is_any");
        in      fn tm => (dest tm; true) handle Fail _ => false
        end
);
```

SML

```
fun mk_any (pattern : TERM, fun_name : string) : TERM list -> TERM = (
	let	val (f, args) = strip_app pattern;
		val arity_s = string_of_int (length args);
		val ftys = map type_of args;
		fun match_arg_tys i (aty :: more_atys) (fty :: more_ftys) = (
			let	val i = type_match1 i aty fty;
			in	match_arg_tys i more_atys more_ftys
			end
		) | match_arg_tys i [] [] = (i
		) | match_arg_tys _ _ _ = (
			fail fun_name 999002 [fn () => arity_s]
		);
		fun match_and_apply tms = (
			let	val atys = map type_of tms;
				val i = match_arg_tys [] atys ftys;
				val f' = inst [] i f;
			in	list_mk_app (f', tms)
			end
		);
	in	match_and_apply
	end
);
fun hd2 (x :: y :: _ : 'a list) : 'a * 'a = (x, y)
|	hd2 _ = fail "hd2" 999003 [];
```

SML

```
local
	val old_thy = get_current_theory_name();
	val _ = open_theory"combin";
	val _ = push_pc"basic_hol1";
in
val mk_o : TERM * TERM -> TERM = (
	let	val mk = mk_any (⌜(t1 : 'b → 'c) o (t2 : 'a → 'b)⌝, "mk_o");
	in	fn (t1, t2) => mk [t1, t2]
	end
);
val dest_o : TERM -> TERM * TERM = (
	let	val dest = dest_any (⌜(t1 : 'b → 'c) o (t2 : 'a → 'b)⌝, "mk_o");
	in	hd2 o dest
	end
);
val is_o : TERM -> bool = is_any ⌜(t1 : 'b → 'c) o (t2 : 'a → 'b)⌝;
val _ = open_theory old_thy;
end;
```

3 PROVING MORPHISMHOOD

3.1 Representing λ-abstractions using first-order combinators

3.1.1 The approach: a rewrite system

We assume given a set of unary operators, binary operators and parametrized operators (such x^n viewed as an operator on x parametrized by n) that are primitive morphisms in some concrete category of interest. We expect the projections $\pi_i : X_1 \times X_2 \to X_i$ to be included amongst the unary operators. We also assume give some set of constant elements of selected objects in the category.

We want to convert a λ-abstraction whose body is a first-order formula built using the given operators, constants and the pairing operator $\_, \_)$ into an equivalent function expressed using the combinators of a category with binary products. We do this using the following rewrite system, where V denotes a *variable structure*, i.e., V is a pattern formed from variables using pairing (such that each free variable of V appears exactly once in V).

$$\begin{array}{rcll}
(\lambda V \bullet x) & \leadsto & \pi_x^V & \text{if } x \in \mathsf{frees}(V) \\
(\lambda V \bullet y) & \leadsto & \mathsf{K}\, y & \text{if } y \notin \mathsf{frees}(V) \\
(\lambda V \bullet c) & \leadsto & \mathsf{K}\, c & \text{if } c \in \mathsf{Constant} \\
(\lambda V \bullet (t_1, t_2)) & \leadsto & \langle(\lambda V \bullet t_1), (\lambda V \bullet t_2)\rangle & \\
(\lambda V \bullet f\, t) & \leadsto & f \circ (\lambda V \bullet t) & \text{if } f \in \mathsf{Unary} \\
(\lambda V \bullet g\, t_1\, t_2) & \leadsto & \mathsf{Uncurry}\, g \circ \langle(\lambda V \bullet t_1), (\lambda V \bullet t_2)\rangle & \text{if } g \in \mathsf{Binary} \\
(\lambda V \bullet h\, t\, p) & \leadsto & (\lambda x \bullet h\, x\, p) \circ (\lambda V \bullet t) & \text{if } h \in \mathsf{Parametrized}
\end{array}$$

Here, if V is a varstruct with a free occurrence of the variable x, π_x^V denotes the combination of projections which extracts x. For example $\pi_x^{((z,x),y)}$ is $\pi_2 \circ \pi_1$. As a special case, $\pi_x^x = \mathsf{I}$ and we may simplify $f \circ I$ to f.

3.1.2 Implementation

We prove template theorems that support the various clauses of the rewrite system.

SML

```
local
        val old_thy = get_current_theory_name();
        val _ = open_theory"combin";
        val _ = push_pc"basic_hol1";
in

val i_rule_thm = snd ("i_rule_thm", (
set_goal([], ⌜(λx• x) = CombI⌝);
a(rewrite_tac [get_spec⌜CombI⌝]);
pop_thm()
));
```

SML

```
val o_i_rule_thm = snd ("o_i_rule_thm", (
set_goal([], ⌜∀f•f o CombI = f⌝);
a(rewrite_tac [get_spec⌜CombI⌝, get_spec⌜$o⌝]);
pop_thm()
));
```

SML

```
val k_rule_thm = snd ("k_rule_thm", (
set_goal([], ⌜∀c• (λx• c) = CombK c⌝);
a(rewrite_tac [get_spec⌜CombK⌝]);
pop_thm()
));
```

SML

```
val unary_rule_thm = snd ("unary_rule_thm", (
set_goal([], ⌜ ∀f t• (λx•f (t x)) = f o t ⌝);
a(rewrite_tac[o_def]);
pop_thm()
));
```

SML

```
val pair_rule_thm = snd ("pair_rule_thm", (
set_goal([], ⌜ ∀s t• (λx•(s x, t x)) = Pair(s, t)⌝);
a(rewrite_tac[pair_def, o_def, uncurry_def]);
pop_thm()
));
```

SML

```
val binary_rule_thm = snd ("binary_rule_thm", (
set_goal([], ⌜ ∀f s t• (λx•f (s x) (t x)) = Uncurry f o Pair(s, t)⌝);
a(rewrite_tac[pair_def, o_def, uncurry_def]);
pop_thm()
));
```

SML

```
val binary_rule_thm1 = snd ("binary_rule_thm1", (
set_goal([], ⌜ ∀f c t• (λx•f c (t x)) = Uncurry f o Pair ((λx•c), t)⌝);
a(rewrite_tac[pair_def, o_def, uncurry_def]);
pop_thm()
));
```

SML

```
val parametrized_rule_thm = snd ("parametrized_rule_thm", (
set_goal([], ⌜ ∀f s p• (λx•f (s x) p) = (λx•f x p) o s⌝);
a(rewrite_tac[o_def]);
pop_thm()
));
```

SML

```
val η_expand_thm : THM = prove_rule[]⌜∀f• f = λz• f z⌝;

val _ = pop_pc();
val _ = open_theory old_thy;
end (* of local ... in ... end *);
```

When we instantiate the template theorems, we want to rename type variables to avoid capture, we use the following utility to help with this.

SML

```
fun list_string_variant (avoid : string list) (ss : string list) : string list = (
	let	fun aux (s, (av, res)) = (
			let	val s' = string_variant av s;
			in	(s'::av, s'::res)
			end
		);
	in	rev(snd (revfold aux ss (avoid, [])))
	end
);
```

The derived rule *gen_∀_elim* is ∀-elimination combined with renaming of type variables to avoid capture.

SML

```
fun gen_∀_elim (tm : TERM) (thm : THM) : THM = (
	let	val tm_tyvs = term_tyvars tm;
		val (asms, conc) = dest_thm thm;
		val thm_tyvs = term_tyvars (mk_list(conc::asms));
		val thm_tyvs' = list_string_variant tm_tyvs thm_tyvs;
		val thm' = inst_type_rule (combine (map mk_vartype thm_tyvs') (map mk_vartype thm_t
	in	∀_elim tm thm'
	end
);
```

The derived rule *all_∀_intro1* gives the universal closure of a theorem but leaving the free variables of a specified term that are not included in a supplied list of "pattern variables" free.

SML

```
fun all_∀_intro1 (pat_vars : TERM list) (tm : TERM) (thm : THM) : THM = (
	let	val fvs = frees tm diff pat_vars;
		val bvs = thm_frees thm diff fvs;
	in	list_∀_intro bvs thm
	end
);
```

Now *morphism_conv* implements our rewrite system.

SML

```
fun morphism_conv
        {unary : TERM list, binary : TERM list, parametrized : TERM list, pattern_vars : TERM list}
        : CONV = (
        let     val unary_thms = map (fn t => all_∀_intro1 pattern_vars t (gen_∀_elim t unary_rule_th
                        unary;
                val binary_thms = map (fn t => all_∀_intro1 pattern_vars t (gen_∀_elim t binary_rule_
                        binary;
                val binary_thms1 = map (fn t => all_∀_intro1 pattern_vars t (gen_∀_elim t binary_rule
                        binary;
                val parametrized_thms = map (switch gen_∀_elim parametrized_rule_thm)
                        parametrized;
                val i_conv = simple_eq_match_conv i_rule_thm;
                val k_conv = simple_eq_match_conv k_rule_thm;
                val pair_conv = simple_ho_eq_match_conv pair_rule_thm;
                val unary_conv = FIRST_C (map simple_ho_eq_match_conv1 unary_thms)
                        handle Fail _ => fail_conv;
                val binary_conv = FIRST_C (map simple_ho_eq_match_conv1 (binary_thms @ binary_th
                        handle Fail _ => fail_conv;
                val parametrized_conv = FIRST_C (map simple_ho_eq_match_conv1 parametrized_thms)
                        handle Fail _ => fail_conv;
                val simp_conv = simple_eq_match_conv o_i_rule_thm;
                val rec rec_conv = (fn t =>
                        ((i_conv ORELSE_C
                        k_conv ORELSE_C
                        (pair_conv THEN_C RAND_C(RANDS_C(TRY_C rec_conv))) ORELSE_C
                        (unary_conv THEN_TRY_C RIGHT_C rec_conv) ORELSE_C
                        (binary_conv THEN_C RIGHT_C (RAND_C(RANDS_C (TRY_C rec_conv))))
                        (parametrized_conv THEN_C RIGHT_C (TRY_C rec_conv)))
                                AND_OR_C simp_conv) t
                );
        in      λ_unpair_conv AND_OR_C rec_conv
        end
);
```

3.2 Moprhismhood Tactic

Now we build the basic morphismhood tactic. It expects a goal of the form $f \in (X, Y)$ *Morphism*. The tactic begins by η-expanding f if it not already an abstraction.

SML

```

val η_expand_conv : CONV = (fn tm => (
        if      is_λ tm
        then    fail_conv
```

```
        else    simple_eq_match_conv η_expand_thm) tm);

```

The theorem is parametrized by a list of theorems that are used as an initial set of rewrite rules. For convenience, we convert any paired abstractions in these theorems into simple abstractions, which makes them more general as rewrite rules.

SML

```

val unpair_rewrite_tac : THM list -> TACTIC =
        rewrite_tac o map (conv_rule (TRY_C (MAP_C λ_unpair_conv)));
```

Now the tactic. After the η-expansion and rewriting discussed above, it converts the function into combinator form. It then goes through a cycle of backchaining with implicative facts applying the supplied tactic to guess existential witnesses, then stripping and rewriting with the basic facts.

SML

```
fun basic_morphism_tac
        {
                unary : TERM list,
                binary : TERM list,
                parametrized : TERM list,
                pattern_vars : TERM list,
                facts : THM list,
                witness_tac : TACTIC} : THM list -> TACTIC = (
        let     val m_conv = morphism_conv {
                        unary = unary,
                        binary = binary,
                        parametrized = parametrized,
                        pattern_vars = pattern_vars};
                val is_rule = is_⇒ o snd o strip_∀ o concl;
                val rule_thms = facts drop (not o is_rule);
                val axiom_thms = facts drop is_rule;
        in      fn rw_thms =>
                        TRY (conv_tac (LEFT_C η_expand_conv))
                THEN TRY (unpair_rewrite_tac rw_thms)
                THEN conv_tac (LEFT_C m_conv)
                THEN            (REPEAT o CHANGED_T) (
                                (TRY o bc_tac) rule_thms
                        THEN TRY witness_tac
                        THEN REPEAT strip_tac
                        THEN (TRY o rewrite_tac) axiom_thms)
        end
);
```

The following constructs witnesses to objecthood using a supplied list of object constructors based on the type of the desired witness. Each object constructor is given with a list of type variables that are not to be instantiated in the search for a witness.

SML

```
fun object_by_type (ocs : (string list * TERM) list) : TYPE -> TERM = (
	let	fun preprocess acc [] = acc
		|	preprocess acc ((tvs, oc) :: more) = (
			let	val rev_tys = rev(strip_→_type (type_of oc));
				val res_ty = hd (rev_tys);
				val tysubs0 = map (fn tv => (mk_vartype tv, mk_vartype tv)) tvs;
				val arg_tys = rev (tl rev_tys);
			in	preprocess ((res_ty, (oc, tysubs0, arg_tys)) :: acc) more
			end
		);
		val table = preprocess [] ocs;
		fun solve [] ty = fail "object_by_type" 1005 []
		|	solve ((res_ty, (oc, tysubs0, arg_tys)) :: more) ty = (
			let	val recur = solve table;
				val tysubs = type_match1 tysubs0 ty res_ty;
				val args = map (recur o inst_type tysubs) arg_tys;
				val ioc = inst [] tysubs oc;
			in	list_mk_app(ioc, args)
			end	handle Fail _ => solve more ty
		);
	in	solve table
	end
);
```

In the following, the list of strings with each object constructor is a list of type variables that are not to be instantiated when matching with this constructor. Typically these would be type variables appearing in the type of something which is an object by dint of an assumption of the goal.

SML

```
fun ∃_object_by_type_tac (ocs : (string list * TERM) list) : TACTIC = (
	let	val witness_by_type = object_by_type ocs;
	in	fn gl as (_, conc) =>
		let	val (x, _) = dest_simple_∃ conc;
		in	(simple_∃_tac o witness_by_type o type_of) x gl
		end
	end
);
(*
```

The following function that extracts lists of known unary, binary and parametrized morphisms and a list of known objects from a list of theorems. The patterns have the form (v, t) where v is a variable and t is a term containing a free occurrence of v to be matched with the conclusion of a theorem. If there is a match in the binary pattern, for example, the appropriate instanve of v is added to the list of binary morphisms.

SML

```
*)
fun analyse_morphism_thms
	{object_pat : TERM, unary_pat : TERM, binary_pat : TERM, parametrized_pat : TERM}
	: THM list ->
		{unary : TERM list, binary : TERM list, parametrized : TERM list} *
			(string list * TERM) list = (
	let	fun dp p = dest_pair p handle Fail _ => (mk_t, mk_t);
		val (object_v, object_p) = dp object_pat;
		val (unary_v, unary_p) = dp unary_pat;
		val (binary_v, binary_p) = dp binary_pat;
		val (parametrized_v, parametrized_p) = dp parametrized_pat;
		fun aux (accs as (acc_u, acc_b, acc_p, acc_o))
			((thm :: more)) = (
			let	val tm = (snd o strip_∀ o concl) thm;
			in	let	val (tym, tmm) = term_match tm binary_p;
					val bin = subst tmm (inst [] tym binary_v);
				in	aux (acc_u, bin::acc_b, acc_p, acc_o) more
				end	handle Fail _ =>
				let	val (tym, tmm) = term_match tm parametrized_p;
					val par = subst tmm (inst [] tym parametrized_v);
				in	aux (acc_u, acc_b, par::acc_p, acc_o) more
				end	handle Fail _ =>
				let	val (tym, tmm) = term_match tm unary_p;
					val un = subst tmm (inst [] tym unary_v);
				in	aux (un::acc_u, acc_b, acc_p, acc_o) more
				end	handle Fail _ =>
				let	val (tym, tmm) = term_match tm object_p;
					val ob = subst tmm (inst [] tym object_v);
					val tvs = (list_cup o map term_tyvars o asms) thm;
				in	aux (acc_u, acc_b, acc_p, (tvs, ob)::acc_o) more
				end	handle Fail _ => aux accs more
			end
		) | aux accs [] = accs;
	in	fn thms => (
			let	val (ul, bl, pl, ol) = aux ([], [], [], []) thms;
			in	({unary = ul, binary = bl, parametrized = pl}, ol)
			end
		)
	end
);
(*
```

The following gives a standard way of constructing the parameters for the morphismhood tactic.